第1章

食わず嫌いのための「演算回路設計」のススメ

——HDLの特徴をうまく生かして設計していこう

小林芳直

最近では，HDL（ハードウェア記述言語）を使って機能を記述し，論理合成ツールによって回路を実現することがあたりまえのようになってきている．HDLによる設計には，基本的な回路のまちがいを除去しながら効率良く設計できるというメリットがある．その一方で，思いどおりの（あるいは要求された）回路規模や性能を実現するために，回路図を頭に思い浮かべて人手でチューンアップしなければならない場合もある．（編集部）

HDL（hardware description language）は，ハードウェアの機能を記述するために使われる言語です．言語の文法にのっとって正しく機能を記述し，そのプログラムを論理合成ツールに通すことで，所望する機能どおりに動作する回路を作ることができます．また，HDLのソース・ファイルは設計データとして使われるだけでなく，そのまま仕様書の一部として扱うこともできます．きちんとした仕様書ができていなくても機能の概略が決まっていれば，外枠（入出力のピン）だけを指定して，設計を進めることもできます．

●“基本的なまちがい”のない回路ができ上がる

ソフトウェア設計でいうと，回路図に対応するのがアセンブリ言語で，HDLに対応するのがC言語です．回路の大部分は，HDLで機能を記述することで実現できます．また，ツールは回路設計者が気づかないようなまちがいをきちんと指摘してくれます．

ASICやFPGAなどの高集積LSIには，軽く数十万ゲートを超す回路素子が搭載されています．回路図でこの一つ一つの回路素子と配線を描いていくよりも，機能をHDLで記述して，ツールに回路を合成させたほうが，基本的なまちがいが少なくなります．また，回路の見通しや可読性も良くなります．

例えば，回路図で回路を記述するときに，回路素子の使いかたを誤ってしまうかもしれません．また，出力が短絡（ショート）している，負荷が重すぎるといったミスも起こりえます．HDLがなかったころは，こうした設計ルールのチェック（DRC：design rule check）をLSI設計者自身が行っており，大きな負担となっていました（図1）．一方，論理合成ツールを利用すると，DRCが自動的に実行されて，基本的な誤りのない回路を作ることができます．

加えて，HDLを利用することでテクノロジ（製造プロセス）の違いを吸収することができます．LSIのテクノロジが変われば，使用するプリミティブ・セル（あらかじめ用意されている基本的なゲート回路）の種類もファンアウト数も変わります．設計のたびにこれらのことを考慮して回路を書くのではたまりません．テクノロジが変わっても，回路ライブラリを入れ替えて合成し直せば新しいテクノロジに対応した回路を作れるということも，HDLの大きなメリ



図1
DRCにかかる設計者の労力を軽減
なにはともあれ，DRCに費やす設計者の労力が減ったのは喜ばしい．

ットです．

● 文法エラーをおそれる必要はない

一般に，ASICやFPGAの設計にはVerilog HDLやVHDLといった言語が使われています．Verilog HDLとVHDLの違いは，ソフトウェアでいうところのJavaとC言語のようなものです．言語が違えば文法なども異なりますが，基本的な考えかたを理解していれば，ほかの言語にも比較的容易に対応できます．

言語というと，「文法など，いろいろと覚えなければならないのでめんどう…」という人もいるかもしれませんが，そこはだいじょうぶ．記述を読むのはツールなので，たとえいいかげんな文法やスペル・ミスがあっても，ていねいに飽きることなく（怒ることもなく…）エラーを見つけて報告してくれます（まちがい探しは人間よりツールのほうがじょうずなので，根をつめてソースの完成度を上げるより，ツールで何回もチェックをかけたほうが効率的）．

回路の設計では，デバッグのことも考えておく必要があります．デバッグというからには，回路の中の悪いところを直さなければなりません．そのために，まずは不ぐあいがあることを見つけて，問題箇所を特定するという作業が必要になります．この作業に，かなり多くの時間を費やします．

一般には，回路の細かいタイミングまで指定して設計してあれば，デバッグは比較的簡単になります．怪しそうな信号をモニタして，その信号が設計者の意図どおり動作していて，かつ動作に不ぐあいがあるのなら，仕様書のミスといえます．一方，設計者の意図どおりに動作していなければ，それは設計ミスです．この仕様書のミスと設計ミスをすみやかに区別できるようにするしかけ（モニタ・ポイント）を作っておくと，トータルの生産性が上がります．

と，設計する前からあれこれ考えていてもしかたがないので，まずは回路設計に必要なツール（ソフトウェア）をそろえて，気楽にソース・コードを書いて回路を合成してみましょう．最近では，FPGA向けの設計ツールの機能限定版が各ベンダのWebサイトから無償でダウンロードできます．また，各社のWebサイトにはチュートリアルやアプリケーション・ノートなどが用意されています．これらもダウンロードして，どんどん読んでいきましょう．

回路設計は，やってみれば（それほど）難しくありません．たとえ迷っても，設計の手助けになる資料は簡単にかつ大量に手に入る時代なのですから…．

もちろん，文法をマスタしただけで「当初の予定どおりに動く回路」が作れるとはかぎりません．動作をしっかりシミュレーションで確認して，さらに実機でも確認しましょう．製品の想定されるライフ・サイクルの中でバグが出なければ，そこで初めて「設計完了！」と言えます．

● お互いの弱点をカバーしながら設計しよう

ここまでの話だと，「HDLは高級言語なので，設計者は機能だけを記述すれば，後は論理合成ツールが動作する回路をはき出してくれるのではないか」と思われることでしょう．この考えは大筋正しいのですが，残念ながらHDLは万能ではありません．現実には，HDL（と論理合成ツール）は“体力勝負”のところでしかうまく働いてくれません（図2）．例えば，非常に大量の配線を単純に正確につなぐことは得意なのですが，細かなラッチの動作，順序制御，排他制御，機能の分割，複数の機能の協調動作といったシステム設計でいちばん難しいところはなかなか対応してくれません．これらを担うのは，結局のところ設計者なのです．

例えば，24ビット入力で，そのうちの12ビット以上が‘1’ならば出力が‘1’になるという回路を考えてみましょう．一般的なHDLの記述では，「‘1’になっているビットの数を数えて，それが12以上なら出力を‘1’にする」という処理になります（リスト1）．しかし，気のきいた設計者であれば，Wallaceツリー[注]と比較器を使ってこの回路を

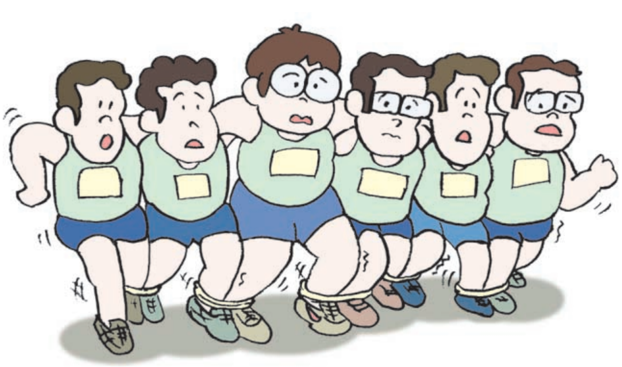


図2　複数機能の協調動作などは苦手…
論理合成ツールは，いろいろと複雑な機能を組み合わせたり，タイミングが交錯するような回路をうまく合成できない．まちがっても「HDLの合成出力だから完ぺきだ」などと過信しないように．

注：複数のバイナリを加算する場合，各けたについてフル・アダーを使って足し合わせていけば，加算すべきバイナリの個数を減らすことができる．複数のフル・アダーを組み合わせて入力の複数のバイナリを二つのバイナリにまで収束させていく回路をWallaceツリーという．Wallaceツリーは，複数のバイナリ・アダーを使う従来の方法と比べて，回路の動作速度が速い．

リスト1　24ビット入力のうち12ビットが'1'かどうかの判定回路

```
function R12( bitz : std_logic_vector(23 downto 0))
  return std_logic is
  variable BitAdd : std_logic_vector(4 downto 0);
  variable GTE12  : std_logic;
  begin
    BitAdd := ( "0000" & bitz(0))
            + ( "0000" & bitz(1))
            + ( "0000" & bitz(2))
            + ( "0000" & bitz(3))
            + ( "0000" & bitz(4))
            + ( "0000" & bitz(5))
            + ( "0000" & bitz(6))
            + ( "0000" & bitz(7))
            + ( "0000" & bitz(8))
            + ( "0000" & bitz(9))
            + ( "0000" & bitz(10))
            + ( "0000" & bitz(11))
            + ( "0000" & bitz(12))
            + ( "0000" & bitz(13))
            + ( "0000" & bitz(14))
            + ( "0000" & bitz(15))
            + ( "0000" & bitz(16))
            + ( "0000" & bitz(17))
            + ( "0000" & bitz(18))
            + ( "0000" & bitz(19))
            + ( "0000" & bitz(20))
            + ( "0000" & bitz(21))
            + ( "0000" & bitz(22))
            + ( "0000" & bitz(23));
    GTE12 := BitAdd(4) or (BitAdd(3) and  BitAdd(2));
  return GTE12;
end R12;
```

作ってしまうでしょう(**リスト2**)．**リスト1**では5ビット・バイナリ・アダーを23個用いているのに対して，**リスト2**では10個のフル・アダーと8個の3ビット・バイナリ・アダーで実現されます．回路規模を比べるとほぼ半分です．また，動作速度についても，**リスト1**の回路は80MHz，**リスト2**の回路は140MHzとこれもほぼ2倍です(ターゲット・デバイスは米国Altera社のFPGA「Stratix EP1S10 F484C5」)．

人間であれば最終的に何を出力にするべきかがわかっているので，大胆な構成をとることができます．しかし，論理合成ツールは基本的に1行ずつ読み込んで回路に変換することしかできません．

HDLにはもっと基本的な問題もあります．それは，実現不可能な回路を記述することを許していることです．例えば，VHDLでは"wait 5ns"などと記述できるのですが，これは回路をシミュレーションするための記述であって，回路を合成するための記述ではありません．つまり，このように記述してシミュレーションできたとしても，そのとおりに動作する回路を合成することはできないのです．

HDLの良さを引き出しながら，弱点は設計者がカバーするような使いかたをしていただければと思います．

リスト2　Wallaceツリーと比較器で構成

ちょっと気のきいた設計者なら，こういった回路を書くだろう．リスト1の回路と比べて，回路規模はほぼ半分，動作速度はほぼ2倍になる．

```
function WallaceTree( pins : std_logic_vector
                                        (23 downto 0))
  return std_logic_vector is
  variable FA0 : std_logic_vector(1 downto 0);
  variable FA1 : std_logic_vector(1 downto 0);
  variable FA2 : std_logic_vector(1 downto 0);
  variable FA3 : std_logic_vector(1 downto 0);
  variable FA4 : std_logic_vector(1 downto 0);
  variable FA5 : std_logic_vector(1 downto 0);
  variable FA6 : std_logic_vector(1 downto 0);
  variable FA7 : std_logic_vector(1 downto 0);
  variable Add0: std_logic_vector(2 downto 0);
  variable Add1: std_logic_vector(2 downto 0);
  variable Add2: std_logic_vector(2 downto 0);
  variable Add3: std_logic_vector(2 downto 0);
  variable Tree: std_logic_vector(11 downto 0);
  begin
    FA7  := FullAdder(pins(23 downto 21));
    FA6  := FullAdder(pins(20 downto 18));
    FA5  := FullAdder(pins(17 downto 15));
    FA4  := FullAdder(pins(14 downto 12));
    FA3  := FullAdder(pins(11 downto  9));
    FA2  := FullAdder(pins( 8 downto  6));
    FA1  := FullAdder(pins( 5 downto  3));
    FA0  := FullAdder(pins( 2 downto  0));
    Add3 := ('0' & FA7)
          + ('0' & FA6);
    Add2 := ('0' & FA5)
          + ('0' & FA4);
    Add1 := ('0' & FA3)
          + ('0' & FA2);
    Add0 := ('0' & FA1)
          + ('0' & FA0);
    Tree := Add3 & Add2 & Add1 & Add0;
  return Tree;
end WallaceTree;

function Over12(Tree : std_logic_vector
                                        (11 downto 0))
  return std_logic is
  variable Add4   : std_logic_vector(2 downto 0);
  variable Add5   : std_logic_vector(2 downto 0);
  variable FA6    : std_logic_vector(1 downto 0);
  variable FA7    : std_logic_vector(1 downto 0);
  variable Add8   : std_logic_vector(2 downto 0);
  variable Add9   : std_logic_vector(2 downto 0);
  variable MJ10   : std_logic;
  variable Toggle : std_logic;
  begin
    Add4 := ('0' & Tree(10 downto  9))
          + ('0' & Tree( 7 downto  6));
    Add5 := ('0' & Tree( 4 downto  3))
          + ('0' & Tree( 1 downto  0));
    FA6  := FullAdder(Tree(11) & Tree(8) & Add4(2));
    FA7  := FullAdder(Tree(5) & Tree(2) & Add5(2));
    Add8 := ('0' & Add4(1 downto 0))
          + ('0' & Add5(1 downto 0));
    Add9 := ('0' & FA6)
          + ('0' & FA7);
    MJ10 := Majority(Add9(0) & Add8(2)
                           & (Add8(1) or Add8(0)));
    Toggle := Add9(2) or ( Add9(1) and MJ10);
  return Toggle;
end Over12;
```

こばやし・よしなお
(株)京セラ　ディスプレイ研究所